Database Languages and their Compilers

Prof. Dr. Torsten Grust

Database Systems Research Group
U Tübingen

Winter 2010

0 Introduction to Query Compilation

0.1 Welcome!

You will have a hard time . . .

. . . to find a more interesting and fun enterprise in Computer Science than the study of the innards of a **Database Query Processor**.

Milestones:

- Dissect *all* phases of an SQL query processor;
- discuss the necessary theoretical toolbox;
- provide the details needed to get an actual implementation going.

In some sense this is a course on **Compiler Construction**, yet the input language and the generated code might appear strange:

Sample Input (SQL):

$$\begin{aligned} \texttt{SELECT} &\ \texttt{COUNT}(z), x, y \\ \texttt{FROM} &\ S \\ \texttt{GROUP BY} &\ x, y \end{aligned}$$

Sample Output (Algebraic Program):

```
1  Insert('S', InlineLoad(['x', 'y', 'z'],
2                         '1|foo|1
3                          1|foo|1
4                          0|bar|0
5  ',
6                          '|','\n'));
7  Commit;
8  Aggr(MScan('S',
9             ['x', 'y', 'z']),
10            [x, y],
11            [cnt = count()]);
```

0.2 Administrativa

- **Lecture time slot** (OK everybody?):
  - Wednesday, 10:15–11:45, in Sand 6/7 (kleiner Hörsaal)
- **Contact me**:
  - Room B 318 (office hours: Thursday, 13:30–15:00)
  - e-mail: `Torsten.Grust@uni-tuebingen.de`
- **Homepage for this course:**
  - `http://www-db.informatik.uni-tuebingen.de/teaching/ws1011/dblang`
- An **oral exam** will conclude this course. Individual appointments.

0.3 Some Remarks on these Slides

This is (partly) new material I am preparing as the semester goes on; I will try to make all slides available prior to the lectures. (Please bear with me.)

> ✎ **Notes**
> We will use non-slide material (software, blackboard) and references to such material will be marked like this. Check **your** notes.

>  **Attention**
> Boxes like this mark material of core interest and it might be advisable to double-check your understanding of the contents.

>  **If you have time . . .**
> . . . you might be interested to follow references in sections like this to gain a deeper understanding of the subject (bedtime reading).

Commercial Systems

Commercial DBMSs[1] sometimes come with interesting query processing technology. These boxes try to point that out.

Caveat

Material in these sections might be tricky to understand (e.g., an algorithm that might not behave as you would guess on first sight)

Woops!

. . . even more tricky stuff! Should not occur too often, though.

[1] If you want a copy of IBM DB2, feel free to ask me.

0.4 Relational Databases and SQL

Let us go through the following to align your and mine vocabulary and explain some fundamental issues.

- Relational database systems represent all user data in tabular structures.
- A specific **relation** (or **table**) S might be depicted like this:

S	x	y	z
	TRUE	'foo'	1
	TRUE	'foo'	1
	FALSE	'bar'	0

  - x, y, and z are the **attributes** of relation S;
  - the possible values in the columns are restricted by the **attribute type**, x has type $\mathbb{B}$ (boolean), we will often write $x :: \mathbb{B}$; here, we also have $y :: \mathbb{S}$ (string) and $z :: \mathbb{N}$ (numeric);

S	x	y	z
	TRUE	'foo'	1
	TRUE	'foo'	1
	FALSE	'bar'	0

- S contains 3 **tuples**, we say $(\texttt{TRUE}, \texttt{'foo'}, 1) \in S$ or even $(x : \texttt{TRUE}, y : \texttt{'foo'}, z : 1) \in S$ if we need to refer to attribute names; all tuples in S are of **tuple type** $(x : \mathbb{B}, y : \mathbb{S}, z : \mathbb{N})$;
- the tuple type $(x : \mathbb{B}, y : \mathbb{S}, z : \mathbb{N})$ is often called the **schema** of S;
- S may contain multiple identical tuples (S is a **multiset** or **bag** of tuples);
- we thus have $S :: bag(x : \mathbb{B}, y : \mathbb{S}, z : \mathbb{N})$, but we will also consider tables $S :: set\ \alpha$ and $S :: list\ \alpha$ with α denoting any tuple type;
- in a database, we assume relation names to be unique; attribute names, however, may be re-used in different relations.

SQL (Structured Query Language) is the de-facto standard query interface offered by RDBMSs.

- SQL comes with language constructs to
  - selectively **access only specific columns** of a table:

```
SELECT  x, y
  FROM  S
```

--→

x	y
TRUE	'foo'
TRUE	'foo'
FALSE	'bar'

  - evaluate **filter predicates** for every tuple:

```
SELECT  *
  FROM  S
 WHERE  z <> 1
```

--→

x	y	z
FALSE	'bar'	0

- **aggregate** table data:

```
SELECT  SUM(z)        -->  | 1 |
  FROM  S                  | 2 |
```

- **join separate tables** on given predicates:

```
SELECT  s.x, s.z, t.z, t.y
  FROM  S AS s, S AS t      -->  |  x     z   z1   y     |
 WHERE  s.z < t.z                |  FALSE 0   1    'foo' |
                                 |  FALSE 0   1    'foo' |
```

- **group tuples** with respect to given attributes and then aggregate over the resulting groups:

```
SELECT  x, y, COUNT(z)      -->  |  x     y      3 |
  FROM  S                        |  TRUE  'foo'  2 |
GROUP BY  x, y                   |  FALSE 'bar'  1 |
```

- **sort** the tuples of a table by specified sort criteria:

```
SELECT  *
  FROM  S
ORDER BY  y ASC
```

--→

x	y	z
FALSE	'bar'	0
TRUE	'foo'	1
TRUE	'foo'	1

- apply **bag operations** (`UNION`, `EXCEPT`, `INTERSECT`):

```
  SELECT  *
    FROM  S
UNION
  SELECT  *
    FROM  S
   WHERE  z = 0
```

--→

x	y	z
TRUE	'foo'	1
TRUE	'foo'	1
FALSE	'bar'	0
FALSE	'bar'	0

- **remove duplicate** tuples in a table:

```
SELECT  DISTINCT *
  FROM  S
```

--→

x	y	z
TRUE	'foo'	1
FALSE	'bar'	0

- evaluate **universally or existentially quantified** predicates:

```
SELECT  *
  FROM  S
 WHERE  FORALL v IN VALUES 1, 2, 3 : v <> z
```

--→

x	y	z
FALSE	'bar'	0

Standard SQL

Standard SQL requires you to write the above query as

```
SELECT  *
  FROM  S
 WHERE  z <> ALL VALUES 1, 2, 3
```

- The result of applying a SQL query to database tables is a table again (SQL is a **closed** language).

- The SQL constructs `DISTINCT` and `ORDER BY` produce tables of type *set* α and *list* α, respectively. (**N.B.** the SQL processor might be completely *bag*-based internally.)

> 🕒 **Relational DBMSs and SQL**
> You will find extensive coverage of relational database management systems (RDBMSs) and the SQL query language on the slides for the lecture "Datenbanksysteme I" (Summer 2010).

Example 0.1

Suppose a cinema keeps track of seat reservations using the following database table `Seats`:

Seats	seat	reserved
	1	FALSE
	2	FALSE
	3	TRUE
	4	FALSE
	5	TRUE
	⋮	⋮

All seats in the cinema are represented in this relation, adjacent seats have adjacent seat numbers (we are ignoring the fact that seats n and $n+1$ might be in different rows).

Use SQL to answer the following questions:

① How many seats are reserved already, how many are still free?

② List all groups of free seats of size 3.

③ List all groups of free seats of size n ($n \geq 1$).

0.5 A Guided Tour through an SQL Query Processor

What follows provides a first insight into the workings of an SQL compiler and query processor. Consider this as a **roadmap for the rest of this course**. We will use the **example data and query** below and proceed step by step, producing a **list of tasks we'll have to tackle** as we go.

Mail	sender	subject	domain
	'Alice'	'I love you'	'.nl'
	'Bob'	'Re: You are fired'	'.uk'
	'Cathy'	'You are fired'	'.uk'
	'Dave'	'I love you'	'.de'
	⋮	⋮	⋮

DNS	domain	country
	'.de'	'Germany'
	'.nl'	'Netherlands'
	'.uk'	'United Kingdom'
	⋮	⋮

```
SELECT  Mail.sender, country
  FROM  Mail, DNS
 WHERE  Mail.domain = DNS.domain
   AND  subject = 'I love you'
```

0.5.1 Character Streams and Tokens

- The SQL processor does not see the query as a whole but consumes user input **character-by-character**:

  `SELECT␣Mail.sender,␣country↰␣FROM ... you'↰`

  - Some characters do not contribute to the meaning of the query but rather define "word" boundaries: `␣` `↰`;
  - it will be helpful for later stages to operate on words rather than characters; identify words and classify them: **tokens**.

`SELECT`	⇢	⟨SELECT⟩
`-42`	⇢	⟨NUM, −42⟩
`Mail`	⇢	⟨ID, 'Mail'⟩

- Input for subsequent stages: **token stream**:

  ⟨SELECT⟩ ⟨ID, 'Mail'⟩ ⟨DOT⟩ ⟨ID, 'sender'⟩ ... ⟨EQ⟩⟨STRING, 'I love you'⟩

0.5.2 Identify Valid SQL Syntax

- Only a subset of token streams represent syntactically valid SQL queries;
  - Streams like $\ldots \langle \texttt{NUM}, -42 \rangle \ \langle \texttt{SELECT} \rangle \ \langle \texttt{FROM} \rangle \ \langle \texttt{FROM} \rangle \ldots$ don't make sense;
- We need **rules** that identify valid token (sub)sequences:

  "A condition is either a boolean or a quantifier or a condition in parentheses."

  "A quantifier either starts with $\langle \texttt{FORALL} \rangle$ *or* $\langle \texttt{EXISTS} \rangle$*, followed by a variable, followed by* $\langle \texttt{IN} \rangle \ldots$*"*
- Define a formal set of rules, a **grammar**, that describes valid token sequences (need **alternatives** and **sequences**):

$$
\begin{array}{lcl}
\mathit{condition} & \rightarrow & \mathit{boolean} \\
& | & \mathit{quantifier} \\
& | & \langle \texttt{(} \rangle \ \mathit{condition} \ \langle \texttt{)} \rangle \\
\\
\mathit{quantifier} & \rightarrow & \langle \texttt{FORALL} \rangle \ \mathit{id} \ \langle \texttt{IN} \rangle \ \mathit{query} \ \langle \texttt{:} \rangle \ \mathit{query} \\
& | & \langle \texttt{EXISTS} \rangle \ \mathit{id} \ \langle \texttt{IN} \rangle \ \mathit{query} \ \langle \texttt{:} \rangle \ \mathit{query}
\end{array}
$$

0.5.3 Resolve the Meaning of Variables and Identifiers

- SQL offers a great deal of freedom when it comes to the usage of variables or attribute names:

```
SELECT  Mail.sender, country
  FROM  Mail, DNS
 WHERE  Mail.domain = DNS.domain
   AND  subject = 'I love you'
```

- It will be our task to **make references explicit** (and complain in case of **ambiguity** etc.), and **come up with attribute names** used in the query result.

- This step saves the user from (a) being too explicit and (b) inventing variable and attribute names:

```
SELECT  x.sender AS sender, y.country AS country
  FROM  Mail AS x, DNS AS y
 WHERE  x.domain = y.domain
   AND  x.subject = 'I love you'
```

0.5.4 Check Types and Schemas

- Syntactical correctness does not imply we're dealing with a meaningful query:
  - ...`WHERE` $42 <$ `'foobar'`...
  - ...`FORALL` v `IN VALUES 'foo','bar'` : $v + 1 < 3$...
- A query with **attribute references** like

$$\begin{array}{rl} \texttt{SELECT} & x.a, x.b + 1 \\ \texttt{FROM} & S\ \texttt{AS}\ x \end{array}$$

  is acceptable only if the tuples in S are of type $(\ldots, a : \alpha, \ldots, b : \mathbb{N}, \ldots)$.
  - The query compiler will have to communicate with the DBMS's **schema manager** to check for schemas and attribute types.

0.5.5 How Could Query Evaluation Look Like?

```
SELECT  Mail.sender, country
  FROM  Mail, DNS
 WHERE  Mail.domain = DNS.domain
   AND  subject = 'I love you'
```

- Something like the pseudo-code program below could evaluate the given query:

```
FOREACH y ∈ DNS DO
    FOREACH x ∈ Mail DO
        IF x.domain = y.domain ∧ x.subject = 'I love you' THEN
            t ← (x.sender, y.country);
            output t;
```

- Execution is mainly driven by **Nested Loops** processing;
- this leads to a complexity of $O(|\texttt{Mail}| \times |\texttt{DNS}|)$ ...
  ... which becomes inacceptable for complex queries.

0.5.6 One Query, Many Programs

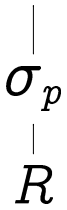
- The program below clearly is **more efficient** yet computes the **same query result**:

> ✎ **Rewrite the Query Program.**

- We have just used two major query optimization techniques:
  - **Join Reordering**, and
  - **Selection Pushdown**.

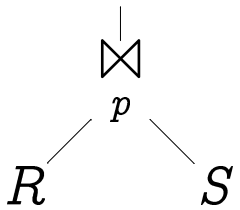
0.5.7 Query Operators

- Typical **program patterns** recur again and again.
  - **Selection** (with respect to predicate[2] p):




```
FOREACH x ∈ R DO
    IF p x THEN
        output x;
```

  - **Join** (with respect to predicate p):

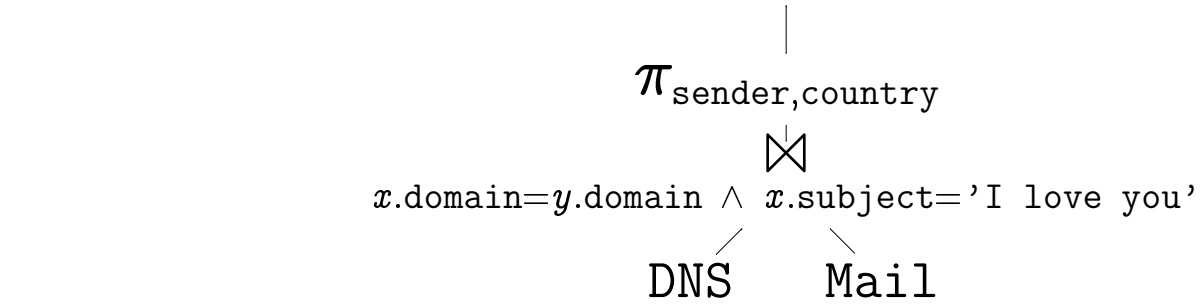



```
FOREACH x ∈ R DO
    FOREACH y ∈ S DO
        IF p (x, y) THEN
            output (x, y);
```

[2] A predicate is a function returning a result of type $\mathbb{B}$, e.g., $\leq :: (\mathbb{N}, \mathbb{N}) \to \mathbb{B}$.

0.5.8 Query Operator Trees and Rewriting

- Query operators are assembled to form **query operator trees**:

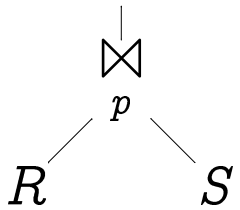


- With the help of operator trees we have a natural way of expressing **query rewriting** steps:

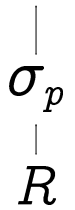
0.5.9 The Cheaper, the Better: Query Cost Models

- How can we guide the query optimizer in making sensible rewriting decisions?
  - Answer: **Rules of Thumb** and **Cost Models**.

- Assign a **cost measure** to each operator estimating its CPU and I/O activity:



$$cost(R \bowtie_p S) = |R| + |R| \times |S| + |R| \times |S| \times s_p$$



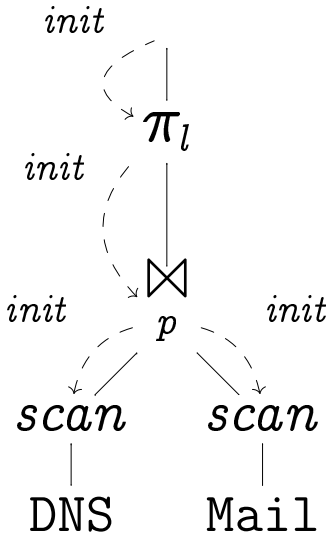
$$cost(\sigma_p(R)) = |R| + |R| \times s_p$$

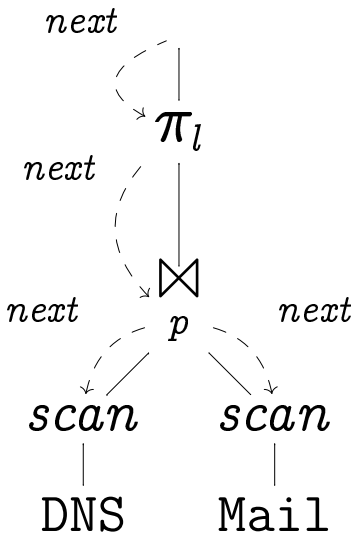
(with $0 \leq s_p \leq 1$ denoting the **selectivity** of predicate p).

> **Cost Estimation is Sufficient**
> We do not care for the actual CPU cycles or number of I/O bursts as long as the cost estimation enables the optimizer to identify the cheaper of two operator trees.

0.5.10 Executing Operator Trees on a Machine

- Each operator implemented by **separate subroutine** in a **query interpreter**;
  - **operators schedule each other** via a simple *init–next–done*? interface:





  - **tuple-at-a-time** scheme; **avoids materialization** of temporary results.

- **Linearize operator tree** in memory; nodes represented by heap addresses; operator invocation $\equiv$ CPU `jmp` instruction.

To conclude this tour, this is a list of **high-level topics** that we will encounter during this course:

① **Syntactal and Semantical Analysis**

- lexing, parsing, parse tree construction
- type checking, variable scoping

② **Query Compilation**

- internal query representation, query calculus and algebra

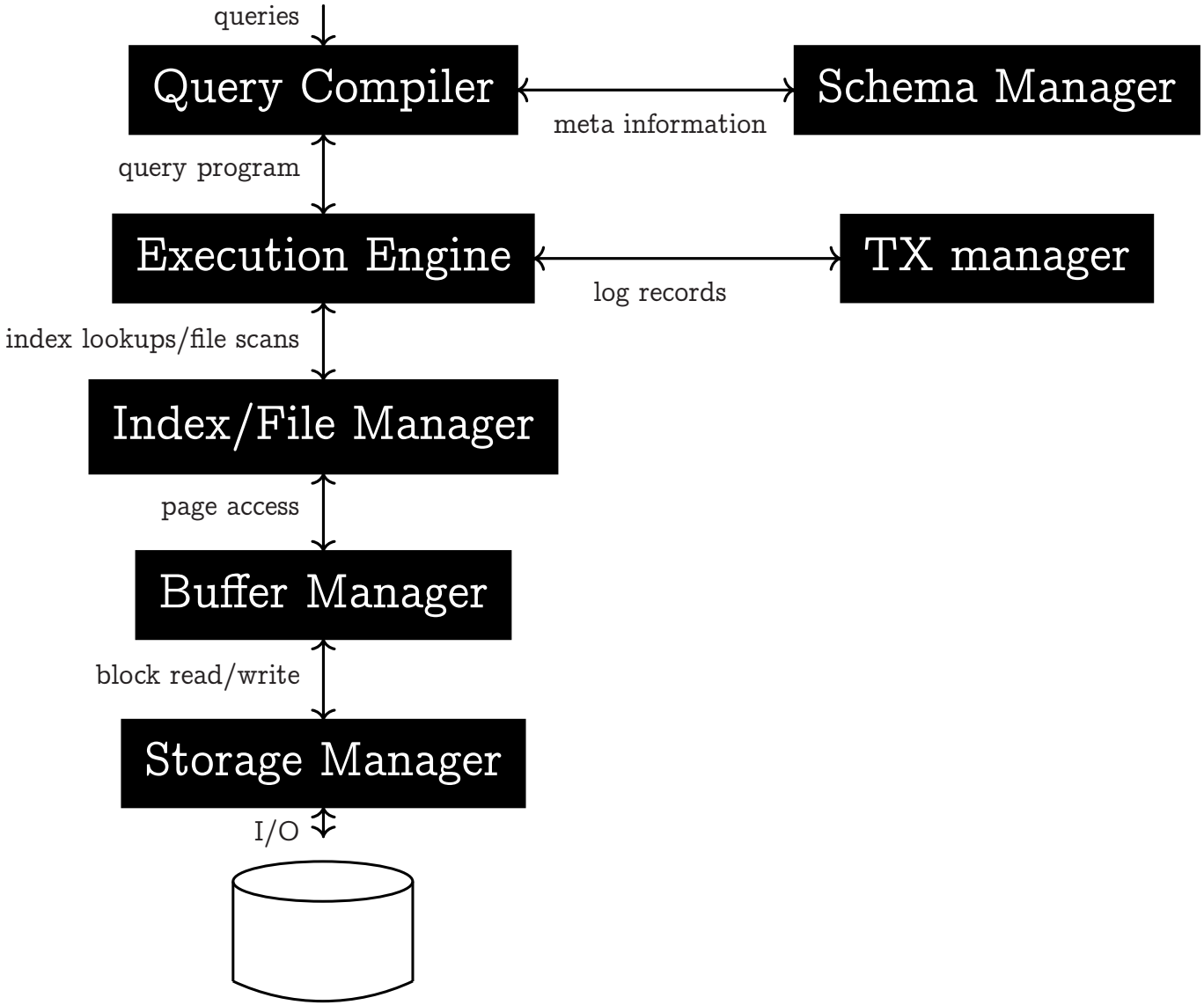
③ **Query Planning and Optimization**

- query rewriting, query costing, searching

④ **Query Execution**

- query interpreter design and implementation

Components of a Typical DBMS



0.6 Recommended Reading

[1] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database System Implementation*, Prentice Hall, New Jersey, 2000. ISBN 0-13-040264-8.

[2] G. Saake and A. Heuer, *Datenbanken: Implementierungstechniken*, MITP Verlag GmbH, 1999. ISBN 3-8266-0513-6.

Contents